love this?
https://dangitgit.com

DANGIT, GIT!

RECIPES FOR GITTING OUT OF A GIT MESS

BY KATIE SYLOR-MILLER
AND JULIA EVANS

what's this?

If you find git confusing, don't worry! You're not alone. People who've been using it every day for years still make mistakes and aren't sure how to fix them. A lot of git commands are confusingly named (why do you create new branches with `git checkout`?) and there are 20 million different ways to do everything.

$ man git reset

In the first and second form, copy entries from <tree-ish> to the index. In the third form, set the current branch head (HEAD) to <commit>, optionally modifying index and working tree to match. The <tree-ish>/<commit> defaults to HEAD in all forms.

actual man page

is this even English?

This zine explains some git fundamentals in plain English, and how to fix a lot of common git mistakes.

By Katie Sylor-Miller & Julia Evans
Website: https://ohshitgit.com
Cover art by Deise Lino

I did something terribly wrong, does git have a magic time machine?

Yes! It's called git reflog and it logs every single thing you do with git so that you can always go back.

Suppose you ran these git commands:

```
git checkout my-cool-branch (1)
git commit -am "add cool feature" (2)
git rebase master (3)
```

Here's what git reflog's output would look like. It shows the most recent actions first:

```
⋮
245fc8d HEAD@{2}  rebase -i (start): (3) checkout master
b623930 HEAD@{3}  commit:            (2) add cool feature
01d7933 HEAD@{4}  checkout:          (1) moving from master
⋮                                        to my-cool-branch
```

If you really regret that rebase and want to go back, here's how:

```
git reset --hard b623930
git reset --hard HEAD@{3}
```

2 ways to refer to that commit before the rebase

I want to undo something from 5 commits ago!

If you made a mistake but want to keep all of the commits since then, git revert is your friend!

`git revert` will create a reverse patch for the changes in a commit and add it as a new commit.

1. Find the commit SHA for the commit you want to undo
2. Run:

```
git revert SHA
```

3. Enter a commit message for the revert commit

Now all of the changes you made in that commit are undone!

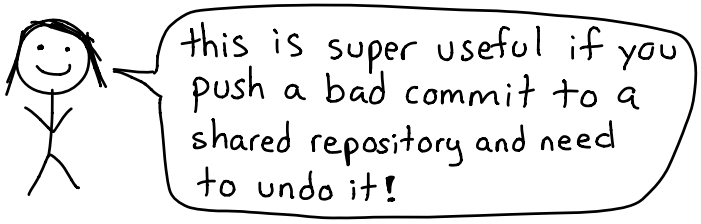


Table of Contents

★ git fundamentals ★

♥ dangit! mistakes & how to fix them ♥

A SHA always refers to the same code

Let's start with some fundamentals! If you understand the basics about how git works, it's WAY easier to fix mistakes. So let's explain what a git commit is!

Every git commit has an id like 3f29abcd233fa, also called a SHA ("Secure Hash Algorithm"). A SHA refers to both:

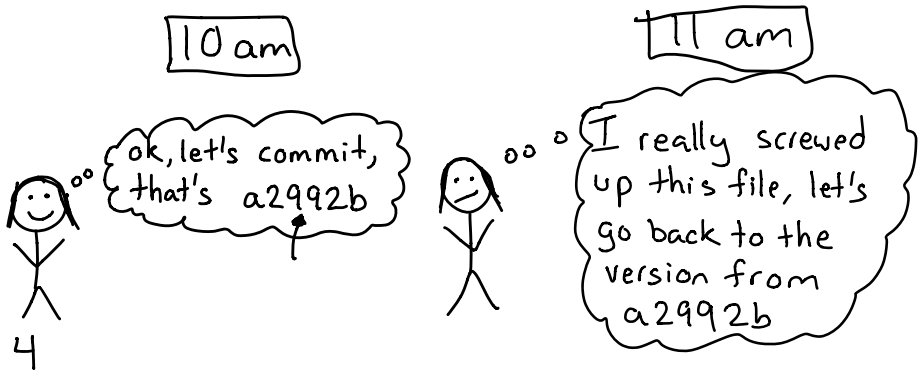
- the changes that were made in that commit ← see them with 'git show'
- a snapshot of the code after that commit was made

No matter how many weird things you do with git, checking out a SHA will always give you the exact same code. It's like saving your game so that you can go back if you die
You can check out a commit like this:

```
git checkout 3f29ab
```

SHAs are long but you can just use the first 6 chars

This makes it way easier to recover from mistakes!



I want to split my commit into 2 commits!

1. Stash any uncommitted changes (so they don't get mixed up with the changes from the commit)

```
git stash
```

2. Undo your most recent commit

```
git reset HEAD^
```

safe: this points your branch at the parent commit but doesn't change any files

3. Use `git add` to pick and choose which files you want to commit and make your new commits!

4. Get your uncommitted changes back

```
git stash pop
```

I started rebasing and now I have 10000000 conflicts to fix!

This can happen when you're rebasing many commits at once.

1. Escape the rebase of doom

```
git rebase --abort
```

2. Find the commit where your branch diverged from master

```
git merge-base master my-branch
```

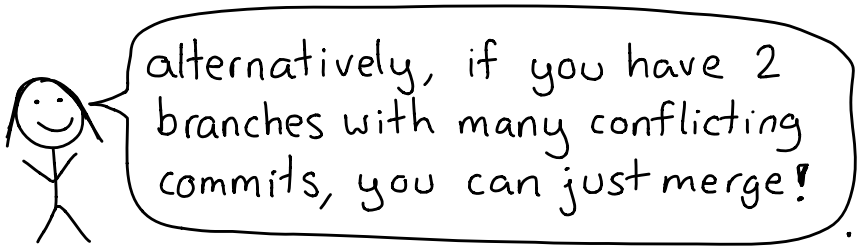
3. Squash all the commits in your branch together

```
git rebase -i SHA_YOU_FOUND
```

output of `git merge-base` goes here

4. Rebase on master

```
git rebase master
```



A branch is a pointer to a commit

A branch in git is a pointer to a commit SHA

master → 2e9fab

awesome-feature → 3bafea

fix-typo → 9a9a9a

Here's some proof! In your favourite git repo, run this command:

```
$ cat .git/refs/heads/master
```

this is just a text file with the commit SHA master points at!

Understanding what a branch is will make it WAY EASIER to fix your branches when they're broken: you just need to figure out how to get your branch to point at the right commit again!

3 main ways to change the commit a branch points to:

- `git commit` will point the branch at the new commit
- `git pull` will point the branch at the same commit as the remote branch
- `git reset COMMIT_SHA` will point the branch at `COMMIT_SHA`

HEAD is the commit you have checked out

In git you always have some commit checked out. HEAD is a pointer to that commit and you'll see HEAD used a lot in this zine. Like a branch, HEAD is just a text file.
Run `cat git/HEAD` to see the current HEAD.

Here are a couple of examples of how to use HEAD:

show the diff for the current commit:

```
git show HEAD
```

UNDO UNDO UNDO UNDO: reset branch to 16 commits ago

```
git reset --hard HEAD~16
```

↖ HEAD~16 means 16 commits ago

show what's changed since 6 commits ago:

```
git diff HEAD~6
```

squash a bunch of commits together

```
git rebase -i HEAD~8
```

Rebasing a branch against itself 8 commits ago lets you squash commits together! (use "fixup")

I committed a file that should be ignored!

Did you accidentally commit a 1.5GB file along with the files you actually wanted to commit? We've all done it.

① Remove the file from Git's index

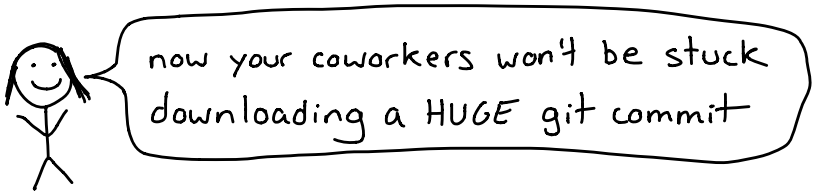
```
git rm --cached FILENAME
```

This is safe: it won't delete the file

② Amend your last commit

```
git commit --amend
```

③ (optional) Edit your .gitignore so it doesn't happen again



I have a merge conflict?!

Suppose you had master checked out and ran `git merge feature-branch`.
When that causes a merge conflict, you'll see something like this in the files with conflicts:

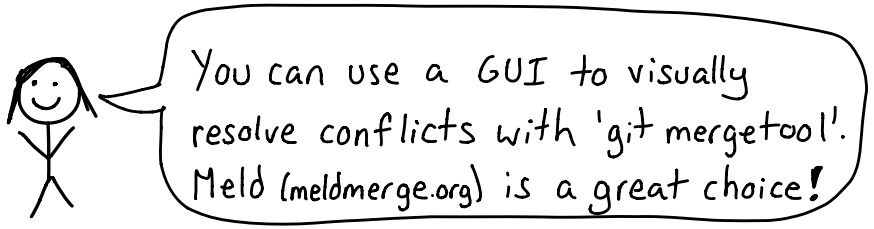
```
<<<<<<< HEAD
if x == 0:
  return false
=======
if y == 6:
  return true
elif x ==0:
  return false
>>>>>>> d34367
```

code from master

code from feature-branch

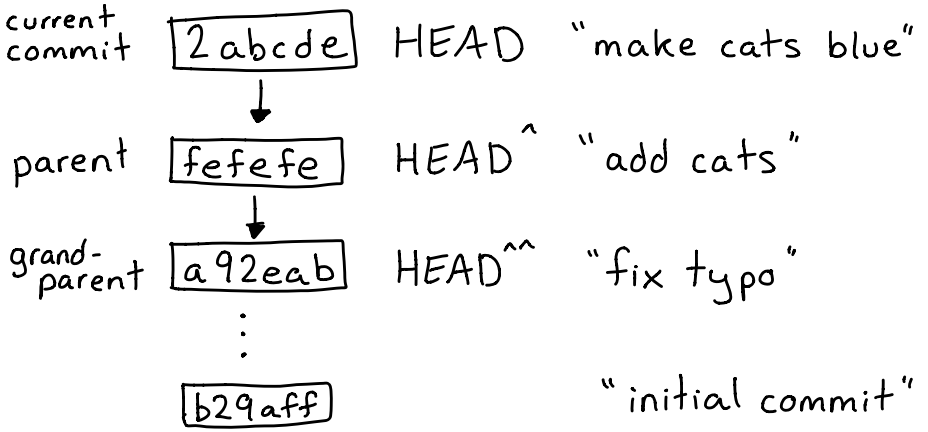
To resolve the conflict:

1. Edit the files to fix the conflict
2. `git add` the fixed files
3. `git diff --check`: check for more conflicts
4. `git commit` when you're done ← or `git rebase --continue` if you're rebasing!



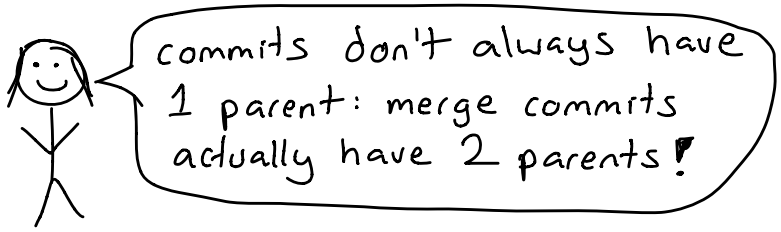
every commit has a parent

Every commit (except the first one!) has a parent commit!
You can think of your git history as looking like this:



HEAD always refers to the current commit you have checked out, and HEAD^ is its parent. So if you want to go look at the code from the previous commit, you can run

```
git checkout HEAD^
```



`git log` shows you all the ancestors of the current commit, all the way back to the initial commit

mistakes you can't fix

Most mistakes you make with git can be fixed. If you've ever committed your code, you can get it back. That's what the rest of this zine is about!

Here are the dangerous git commands: the ones that throw away uncommitted work.

`git reset --hard COMMIT`

1. Throws away uncommitted changes
2. Points current branch at COMMIT

Very useful, but be careful to commit first if you don't want to lose your changes

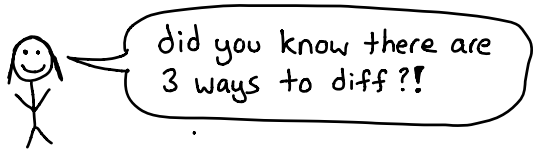
`git clean`

Deletes files that aren't tracked by Git.

`git checkout BRANCH FILE` ← or directory

Replaces FILE with the version from BRANCH. Will overwrite uncommitted changes.

I tried to run a diff but nothing happened?



Suppose you've edited 2 files:

```
$ git status
On branch master
Changes to be committed:

  modified:  staged.txt

Changes not staged for commit:

  modified:  unstaged.txt
```

staged changes (added with 'git add')

unstaged changes

Here are the 3 ways git can show you a diff for these changes:

- `git diff`: unstaged changes
- `git diff --staged`: staged changes
- `git diff HEAD`: staged+unstaged changes

A couple more diff tricks:

- `git diff --stat` gives you a summary of which files were changed & number of added/deleted lines
- `git diff --check` checks for merge conflict markers & whitespace errors

I committed something to master that should have been on a brand new branch!

① Make sure you have master checked out

```
git checkout master
```

② Create the new branch

```
git branch my-new-branch
```

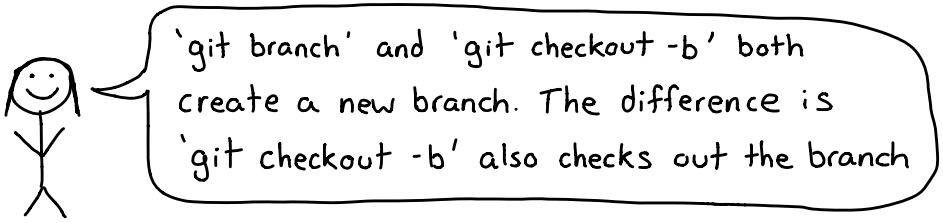
③ Remove the unwanted commit from master

```
git status
git reset --hard HEAD~
```

careful!

④ Check out the new branch!

```
git checkout my-new-branch
```



I need to change the message on my last commit!

No problem! Just run:

```
git commit --amend
```

Then edit the commit message & save!

`git commit --amend` will replace the old commit with a new commit with a new SHA, so you can always go back to the old version if you really need to.

I committed but I need to make one small change!

① Make your change
② Add your files with `git add`
③ Run:

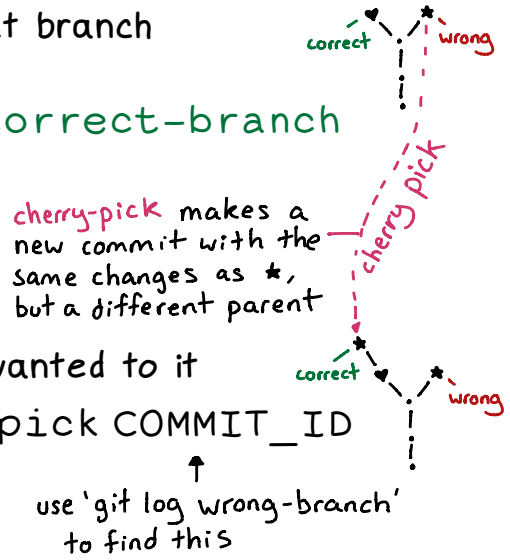
```
git commit --amend --no-edit
```

You can also add a new commit and use `git rebase -i` to squash them but this is about a million times faster.

I accidentally committed to the wrong branch!

① Check out the correct branch

```
git checkout correct-branch
```

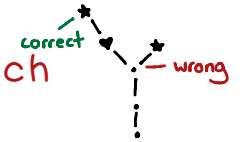


② Add the commit you wanted to it

```
git cherry-pick COMMIT_ID
```

use 'git log wrong-branch' to find this

③ Delete the commit from the wrong branch




```
git checkout wrong-branch
git reset --hard HEAD^
```